Recursion Schemata for NCk

Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, Isabel Oitavem

Recursion schemata for NC^k

Guillaume Bonfante[1], Reinhard Kahle[2], Jean-Yves Marion[1], and Isabel Oitavem[3]

[1] Loria - INPL, 615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy, France, {Jean-Yves.Marion|Guillaume.Bonfante}@loria.fr
[2] CENTRIA and DM, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal, kahle@mat.uc.pt
[3] UNL and CMAF, Universidade de Lisboa, Av. Prof. Gama Pinto, 2, 1649-003 Lisboa, Portugal, isarocha@ptmat.fc.ul.pt

Abstract. We give a recursion-theoretic characterization of the complexity classes NC^k for $k \geq 1$. In the spirit of implicit computational complexity, it uses no explicit bounds in the recursion and also no separation of variables is needed. It is based on three recursion schemes, one corresponds to time (time iteration), one to space allocation (explicit structural recursion) and one to internal computations (mutual in place recursion). This is, to our knowledge, the first exact characterization of the NC^k by function algebra over infinite domains in implicit complexity.

1 Introduction

Since the seminal works of Simmons [19], of Leivant [11,12], of Bellantoni and Cook [3], and of Girard [8], *implicit computational complexity* has provided models over infinite domains of major complexity classes which are independent from the notion of time or of space related to machines.

These studies have nowadays at least two twin directions. The first direction concerns the characterization of complexity classes by means of logics or of recursion schemas. A motivation is to have a mathematical model of resource-bounded computations. The second direction is more practical and aims to analyze and certify resources, which are necessary for a program execution. One of the major challenges here is to capture a broad class of useful programs whose complexity is bounded. There are several approaches [1,10,16,6].

This paper falls in the first direction which can be seen as a guideline for the second approach. We give a recursion-theoretic characterization of each class NC^k by mean of a function algebra INC^k based on tree recursion. We demonstrate that $\mathsf{INC}^k = NC^k$ for $k \geq 1$.

The classes NC^k were firstly described based on circuits. NC^k is the class of functions accepted by uniform boolean circuit families of depth $O(\log^k n)$

Research supported by the project *Teorias e linguagens de programação para computações com recursos limitados* within the *Programa PESSOA 2005/2006* of *GRICES - EGIDE* and partly by the FCT project POCI/MAT/61720/2004 and by DM, FCT-UNL.

and polynomial size with bounded fan-in gates, where n is the length of the input—see [2] or [9]. In [18], Ruzzo identifies NC^k with the classes of languages recognized by alternating Turing machines (in short ATMs) in time $O(\log^k n)$ and space $O(\log n)$.

In fact, the main difficulty in this characterization of NC^k relies on the double constraint about time and space. Other previous characterizations based on tree recursion fail to exactly capture for this reason. In 1998, Leivant [13] characterized NC using a hierarchy of classes RSR, such that $\mathsf{RSR}_k \subseteq \mathrm{NC}^k \subseteq \mathsf{RSR}_{k+2}$ for $k \geq 2$. In the sequence of [4] and [17], this result was refined in [5] by defining term systems T^k such that $T^k \subseteq \mathrm{NC}^k \subseteq T^{k+1}$ for $k \geq 2$. Both approaches are defined in a sorted context, either with safe/normal arguments or with tiered recursion.

We define INC^k as classes of functions, over the tree algebra $\mathbb{T}$, closed under composition and three recursion schemes over $\mathbb{T}$: time iteration, explicit structural recursion and mutual in place recursion. No explicit bounds are used in the schemes and also no separation of variables is needed. The mutual in place recursion scheme, one main point of our contribution, is related to previous work of Leivant and Marion, see [14]. The absence of tiering mechanism is related to [15], so that similar diagonalization argument should be possible.

2 Preliminaries

Let $\mathbb{W}$ be the set of words over $\{0,1\}$. We denote by ϵ the empty word and by $\mathbb{W}_i$ the subset of $\mathbb{W}$ of words of length exactly i. We consider the tree algebra $\mathbb{T}$, generated by three 0-ary constructors $\mathbf{0}, \mathbf{1}, \perp$ and a binary constructor $\star$, in other words, binary trees with leaves are labeled by $\{\mathbf{0}, \mathbf{1}, \perp\}$[1]. $\mathsf{S}(t)$ denotes the size of a tree, $\mathsf{H}(t)$ corresponds to the usual notion of height. We say that a tree t is *perfectly balanced* if it has $2^{\mathsf{H}(t)}$ leaves. All along, 0 serves as false, 1 as true and $\perp$ as the undefined.

Given a non-empty (enumerable) set of variables $\mathcal{X}$, we denote by $\mathbb{T}(\mathcal{X})$ the term-algebra of binary trees whose leaves are labeled by $\mathbf{0}, \mathbf{1}, \perp$ or variables from $\mathcal{X}$. If t, u denote some terms and x is a variable, the term $t[x \leftarrow u]$ denotes the substitution of x by u in t. Then, $t[x \leftarrow u, y \leftarrow v] = t[x \leftarrow u][y \leftarrow v]$. All along, we take care to avoid clashes of variables. When we have a collection I of variable substitutions, we use the notation $t[(x_w \leftarrow u_w)_{w \in I}]$. Again, we will avoid conflicts of variables.

We now define some convenient notations, used extensively all along the paper. Given a set of variables $\mathcal{X} = (x_w)_{w \in \mathbb{W}}$, we define a family of balanced trees that we call tree patterns $(\mathfrak{t}_i)_{i \in \mathbb{N}}$ in $\mathbb{T}(\mathcal{X})$ where each leaf is labeled by a

[1] Actually, we could build trees by means of only two constructors $\star, \mathtt{nil}$ and develop the theory out of this algebra. This choice, though correct, involves much more tedious encodings which we will avoid without changing in essence the theory.

distinct variable:

$$\begin{aligned} \mathfrak{t}_0 &= x_\epsilon \\ \mathfrak{t}_{i+1} &= \mathfrak{t}_i[(x_w \leftarrow x_{0w})_{w\in\mathbb{W}_i}] \star \mathfrak{t}_i[(x_w \leftarrow x_{1w})_{w\in\mathbb{W}_i}] \end{aligned}$$

Observe that the index of a variable of some tree pattern indicates the path from the root to it. For example, $\mathfrak{t}_2 = (x_{00} \star x_{01}) \star (x_{10} \star x_{11})$. The use of the $\mathfrak{t}$'s and substitutions makes notations very short. For instance, $\mathfrak{t}_2[(x_w \leftarrow f_w(x_w))_{w\in\mathbb{W}_2}] = (f_{00}(x_{00}) \star f_{01}(x_{01})) \star (f_{10}(x_{10}) \star f_{11}(x_{11}))$. This notation is particularly useful to define "big-step" recursion equations as in:

$$f((x_{00} \star x_{01}) \star (x_{10} \star x_{11})) = (f(x_{00}) \star f(x_{01})) \star (f(x_{10}) \star f(x_{11}))$$

which we shall note: $f(\mathfrak{t}_2) = \mathfrak{t}_2[(x_w \leftarrow f(x_w))_{w\in\mathbb{W}_2}]$

3 The classes INC^k

Definition 1. *The set of* basic functions *is* $\mathcal{B} = \{\mathbf{0}, \mathbf{1}, \bot, \star, (\pi_i^j)_{i\le j}, \mathsf{cond}, \mathsf{d}_0, \mathsf{d}_1\}$ *where* $\mathbf{0}, \mathbf{1}, \bot$ *and* $\star$ *are the constructors of the algebra* $\mathbb{T}$, d_0 *and* d_1 *are the destructors of* $\mathbb{T}$, cond *is a conditional and* π_i^j *are the projections. Destructors and conditional are defined as follows:*

$$\begin{aligned} \mathsf{d}_0(c) = \mathsf{d}_1(c) = c, \quad & c \in \{\mathbf{0}, \mathbf{1}, \bot\} \\ \mathsf{d}_0(t_0 \star t_1) = t_0, \quad & \mathsf{d}_1(t_0 \star t_1) = t_1, \\ \mathsf{cond}(\mathbf{0}, x_0, x_1, x_\bot, x_\star) = x_0, \quad & \mathsf{cond}(\mathbf{1}, x_0, x_1, x_\bot, x_\star) = x_1, \\ \mathsf{cond}(\bot, x_0, x_1, x_\bot, x_\star) = x_\bot, \quad & \mathsf{cond}(t_0 \star t_1, x_0, x_1, x_\bot, x_\star) = x_\star. \end{aligned}$$

The set of basic functions closed by composition is called the set of explicitly defined functions*. If the output of a function is* $\mathbf{0}, \mathbf{1}$ *or* $\bot$*, then we say that the function is* boolean*. If the definition of a function does not use* $\star$*, the function is said to be* $\star$*-free. As a shorthand notation, we use* $\mathsf{d}_{b_1\cdots b_k}$ *for the function* $\mathsf{d}_{b_k} \circ \cdots \circ \mathsf{d}_{b_1}$.

Definition 2. INC^k *is the closure of the set* $\mathcal{B}$ *under composition, mutual in place recursion (*MIP*), explicit structural recursion (*ESR*), and time iteration (*TI*) for* k.

The mentioned schemes are described below.

To relate functions over words to functions over trees, we encode words of $\mathbb{W}$ by perfectly balanced trees of $\mathbb{T}$. For this, we define $\mathtt{tr}(w)$ as the perfectly balanced tree of height $\lceil \log(|w|) \rceil$ whose leaves read from left to right are the letters of w padded by $\bot$ on the right if necessary.

A function $\phi : \mathbb{W}^n \to \mathbb{W}$ is represented by a function $f \in \mathbb{T}^k \to \mathbb{T}$ iff for all words $w_1, \ldots, w_n$, $f(\mathtt{tr}(w_1), \ldots, \mathtt{tr}(w_n)) = \mathtt{tr}(\phi(w_1, \ldots, w_n))$. Actually, the representation of $\phi(w_1, \ldots, w_n)$ does not need to be canonical, that is the height of the output tree may be greater than $\lceil \log(|\phi(w_1, \ldots, w_n)|) \rceil$.

Theorem 3. *For $k \geq 1$, the set of functions over words represented in INC^k is exactly the set of functions computed by circuits in NC^k.*

The proof of the theorem is a direct consequence of Proposition 13 and Proposition 14 coming in Section 4 and 5.

3.1 Mutual in place recursion

As a shorthand for finite sequences, we use $(\bar{\cdot})$. The notation can be nested such as in $\bar{\sigma}(\bar{u})$ which denotes a sequence $\sigma_1(u_1, \ldots, u_{k_1}), \ldots, \sigma_n(u_1, \ldots, u_{k_n})$.

The first recursion scheme, *mutual in place recursion*, is the key element of our characterization.

Definition 4. *The functions $(f_i)_{i\in I}$ (with the set I finite) are defined by* mutual in place recursion *(*MIP*) if they are defined by a set of equations, with $i, j, l \in I$ and $c \in \{\mathbf{0}, \mathbf{1}, \bot\}$, of the form*

$$f_i(t_0 \star t_1, \bar{u}) = f_j(t_0, \bar{\sigma}_{i,0}(t_0 \star t_1, \bar{u})) \star f_l(t_1, \bar{\sigma}_{i,1}(t_0 \star t_1, \bar{u})) \quad (1)$$

$$f_i(c, \bar{u}) = g_{i,c}(\bar{u}) \quad (2)$$

where $\bar{\sigma}_{i,0}$ and $\bar{\sigma}_{i,1}$ are sequences of $\star$-free explicitely defined functions and the functions $g_{i,c}$ are explicitely defined boolean functions.

Notice that the first argument is shared by the entire set of mutually defined functions as recursion argument. While for the others, copies, switch and visit can be performed freely. As a consequence, for any such function f, one may observe that $f(t, \bar{x})$ is a tree with the exact shape of t but, possibly, with different leaves. Actually, informally, to compute the value corresponding to each leaf, one first runs a transducer using the path to that leaf as input. At the end, one computes the bit by a conditional using the outputs of the transducer as pointers to some bits in the input tree.

Example 5. The following function turns the leaves its argument to some fixed constant $c \in \{\mathbf{0}, \mathbf{1}, \bot\}$:

$$\begin{aligned} \mathsf{const}_c(t_0 \star t_1) &= \mathsf{const}_c(t_0) \star \mathsf{const}_c(t_1) \\ \mathsf{const}_c(c') &= c \qquad\qquad c' \in \{\mathbf{0}, \mathbf{1}, \bot\} \end{aligned}$$

Taking the convention that $b \vee \bot = \bot \vee b = \bot$, one may compute (with MIP-recursion) the bitwise-or of two perfectly balanced trees of common size.

$$\begin{aligned} \mathsf{or}(t_0 \star t_1, u) &= \mathsf{or}(t_0, \mathsf{d}_0(u)) \star \mathsf{or}(t_1, \mathsf{d}_1(u)) \\ \mathsf{or}(\mathbf{0}, u) &= \mathsf{cond}(u, \mathbf{0}, \mathbf{1}, \bot, \bot) \\ \mathsf{or}(\mathbf{1}, u) &= \mathsf{cond}(u, \mathbf{1}, \mathbf{1}, \bot, \bot) \\ \mathsf{or}(\bot, u) &= \bot \end{aligned}$$

Actually, all "bitwise boolean formula" of several balanced trees of the same size can be written in a similar manner.

We now give some closure properties of MIP-definable functions, the first one allows us to define a family of MIP-definable functions in terms of the shorthand notation defined above.

Lemma 6. *We suppose given a (finite) family $(n_i)_{i\in I}$ of integers, and a family $(f_i)_{i\in I}$ of functions satisfying equations of the form:*

$$f_i(\mathfrak{t}_{n_i}, \bar{u}) = \mathfrak{t}_{n_i}[(x_w \leftarrow f_{\mathfrak{p}(i,w)}(x_w, \bar{\sigma}_{i,w}(\bar{u})))_{w\in\mathbb{W}_{n_i}}], \tag{3}$$

$$f_i(\mathfrak{t}_m[(x_w \leftarrow c_w)_{w\in\mathbb{W}_m}], \bar{u}) = \mathfrak{t}_m[(x_w \leftarrow g_{i,w,c_w}(\bar{u}))_{w\in\mathbb{W}_m}], 0 \le m < n_i, \tag{4}$$

where $\mathfrak{p}$ is a finite mapping from $I \times \mathbb{W}$ to I, $c_w \in \{\mathbf{0}, \mathbf{1}, \bot\}$, $\bar{\sigma}_{i,w}$ are vectors of $\star$-free explicitly defined functions, and $(g_{i,w,c_w})_{i\in I, w\in\mathbb{W}, c_w\in\{\mathbf{0},\mathbf{1},\bot\}}$ are explicitely defined boolean functions. Then, the functions $(f_i)_{i\in I}$ are MIP*-definable.*

One may note that the equations above specify the functions only for well balanced trees. Since we only use the Lemma for such trees, we do not care with their values for other inputs given by the proof bellow.

Proof. In an equation such as Equation (3), we call n_i the level of the definition of f_i. The proof is by induction on the maximal level of the functions $N = \max_{i\in I} n_i$. If $N = 1$, then the equations correspond to usual MIP-equations.

Suppose now $N > 1$. For all the indices i such that f_i has level N, we replace its definitional equations by:

$$\begin{aligned}
f_i(t_0 \star t_1, \bar{u}) &= f_{i\bullet 0}(t_0, \bar{u}) \star f_{i\bullet 1}(t_1, \bar{u}) \\
f_{i\bullet w}(t_0 \star t_1, \bar{u}) &= f_{i\bullet w0}(t_0, \bar{u}) \star f_{i\bullet w1}(t_1, \bar{u}), && (1 < |w| < N-1) \\
f_{i\bullet w}(t_0 \star t_1, \bar{u}) &= f_{\mathfrak{p}(i,w0)}(t_0, \bar{\sigma}_{i,w0}(\bar{u})) \star f_{\mathfrak{p}(i,w1)}(t_1, \bar{\sigma}_{i,w1}(\bar{u})), && (|w| = N-1) \\
f_{i\bullet w}(c, \bar{u}) &= g_{i,w,c}(\bar{u}), && (1 \le |w| < N) \\
f_i(c, \bar{u}) &= g_{i,\epsilon,c}(\bar{u})
\end{aligned}$$

where the indices $i \bullet w$ are fresh. One may observe that the level of each of these functions is 1. We end by induction.

The following Lemma is easy to verify:

Lemma 7. *Suppose that $f \in (f_i)_{i\in I}$ is defined by* MIP*-recursion. Then, any function $g(t, \bar{u}) = f(t, \bar{\sigma}(t, \bar{u}))$ where the $\bar{\sigma}$ are $\star$-free explicitly defined functions can be defined by* MIP*-recursion.*

3.2 Explicit structural recursion

The recursion scheme defined here corresponds to the space aspect of functions definable in INC^k. It will be used to construct trees of height $O(\log n)$, see the following Lemma.

Definition 8. Explicit structural recursion *(*ESR*) is the following schema:*

$$\begin{aligned} f(t_0 \star t_1, \bar{u}) &= h(f(t_0, \bar{u}), f(t_1, \bar{u})) \\ f(c, \bar{u}) &= g(c, \bar{u}) \qquad\qquad c \in \{\mathbf{0}, \mathbf{1}, \bot\} \end{aligned}$$

where h and g are explicitely defined.

Lemma 9. *Given two natural numbers α_0 and α_1, there is a function f defined by* ESR *such that for any tree t, $\mathsf{H}(f(t)) = \alpha_1 \mathsf{H}(t) + \alpha_0$.*

Proof. The proof is immediate, taking f defined by explicit structural recursion with $h = h_{\alpha_1}$ and $g = h_{\alpha_0}(\mathbf{1}, \mathbf{1})$ where $h_1(w_0, w_1) = w_0 \star w_1$ and $h_i(w_0, w_1) = h_{i-1}(w_0, w_1) \star h_{i-1}(w_0, w_1)$ for $i > 1$.

3.3 Time iteration

The following scheme allows us to iterate MIP-definable functions. It serves to capture the time aspect of functions definable in NC^k. The scheme depends on the parameter k used for the stratification.

Definition 10. *Given $k \geq 1$, a function f is defined by k-*time iteration *(k-*TI*) from the function h which is* MIP*-definable and the function g if:*

$$\begin{aligned} f(t_1' \star t_1'', t_2, \ldots, t_k, s, \bar{u}) &= h(f(t_1', t_2, \ldots, t_k, s, \bar{u}), \bar{u}) \\ f(c_1, t_2' \star t_2'', t_3, \ldots, t_k, s, \bar{u}) &= f(s, t_2', t_3 \ldots, t_k, s, \bar{u}) \\ &\vdots \\ f(c_1, \ldots, c_{i-1}, t_i' \star t_i'', t_{i+1}, \ldots, t_k, s, \bar{u}) &= f(c_1, \ldots, c_{i-2}, s, t_i', t_{i+1}, \ldots, t_k, s, \bar{u}) \\ &\vdots \\ f(c_1, \ldots, c_k, s, \bar{u}) &= g(s, \bar{u}) \end{aligned}$$

where $c_1, \ldots, c_k \in \{\mathbf{0}, \mathbf{1}, \bot\}$.

Lemma 11. *Given a* MIP*-definable function h, a function g and constants β_1 and β_0, there is a function f defined by k-*TI *such that for all perfectly balanced trees t*

$$f(t, \bar{u}) = \underbrace{h(\ldots h}_{\beta_1(\mathsf{H}(t))^k + \beta_0 \ \mathit{times}} (g(t, \bar{u}), \bar{u}) \ldots).$$

Proof. The proof follows the lines of Lemma 9.

4 Simulation of alternating Turing machines

We introduce alternating random access Turing machines (ARMs) as described in [14] by Leivant, see also [7,18]. An ARM $M = (Q, q_0, \delta)$ consists of a (finite) set

of states Q, one of these, q_0, being the initial state and actions δ to be described now. States are classified as disjunctive or conjunctive, those are called action states, or as accepting, rejecting and reading states. The operational semantics of an ARM, M, is a two stage process: firstly, generating a computation tree; secondly, evaluating that computation tree for the given input. A configuration $K = (q, w_1, w_2)$ consists of a state q and two work-stacks $w_i \in \mathbb{W}$, $i \in \{1, 2\}$. The initial configuration is given by the initial state q_0 of the machine and two empty stacks.

In the first step, one builds a computation tree, a tree whose nodes are configurations. The root of a computation tree is the initial configuration. Then, if the state of a node is an action state, depending on the state and on the bits at the top of the work-stacks, one spawns a pair of successor configurations obtained by pushing/popping letters on the work-stacks. The t-time computation tree is the tree obtained by this process until height t.

Wlog, we assume that for each action state q, one of the two successor configurations of q, let us say the first one, lets the stacks unchanged. And for the second successor configuration, either the first stack or the second one is modified, but not both simultaneously. We write accordingly the transition function δ for action states: $\delta(q, a, b) = (q', q'', \mathsf{pop}_i)$ with $i \in \{1, 2\}$ means that being in state q with top bits being a and b, the first successor configuration has state q' and stacks unchanged, and the second successor has state q'' and pops one letter on stack i. When we write $\delta(q, a, b) = (q', q'', \mathsf{push}_i(c))$, with $i \in \{1, 2\}$ and $c \in \mathbb{W}_1$, it is like above but we push the letter c on the top of the stack i.

The evaluation of a finite computation tree T is obtained as follows. Begining from the leaves of T until its root, one labels each node (q, w_1, w_2) according to:

- if q is a rejecting (resp. accepting) state, then it is labeled by 0 (resp. 1);
- if q is a c, j-reading state ($c = 0, 1, j = 1, 2$), then it is labeled by 0 or 1 according to whether the n'th bit of the input is c, where n is the content read on the j'th stack. If n is too large, the label is $\perp$;
- if q is an action state,
  - if it has zero or one child, it is labeled $\perp$;
  - if it has two children, take the labels of its two children and compute the current label following the convention that $c = (c \vee \perp) = (\perp \vee c) = (c \wedge \perp) = (\perp \wedge c)$ with $c \in \{0, 1, \perp\}$.

The label of a computation tree is the label of the root of the computation tree thus obtained.

We say that the machine works in time $f(n)$ if, for all inputs, the $f(n)$-time tree evaluates to 0 or 1 where n is the size of the input. It works in space $s(n)$ if the size of the stacks are bounded by $s(n)$.

Actually, to relate our function algebra to the NC^k, we say that a function is in $\mathrm{ATM}(O(\log^k n, O(\log n))$, for $k \geq 1$ if it is polynomially bounded and bitwise computed by an ATM working in time $O(\log^k n)$ and space $O(\log n)$.

Theorem 12 (Ruzzo [18]). NC^k *is exactly the set of languages recognized by ARM working in time* $O(\log(n)^k)$ *and space* $O(\log(n))$.

From that, one inclusion (from the right to the left) of our main theorem is a corollary of:

Proposition 13. *Given $k \geq 1$ and constants $\alpha_1, \alpha_0, \beta_1, \beta_0$, any* ARM *working in space $\alpha_1 \log(n) + \alpha_0$ and time $\beta_1 \log^k(n) + \beta_0$, where n is the length of the input, can be simulated in INC^k.*

Proof (sketch). Let us consider such a machine $M = (Q, q_0, \delta)$. Take $d = \lceil \log(|Q|) \rceil$. We attribute to each state in Q a word $w \in \mathbb{W}_d$ taking the convention that the initial state q_0 has encoding $0 \cdots 0$. From now on, the distinction between the state and its associated word is omitted.

Let us consider the encoding of two stacks $s_1 = a_1 a_2 \cdots a_i \in \mathbb{W}$ and $s_2 = b_1 b_2 \cdots b_j \in \mathbb{W}$ of length less or equal than $\alpha_1 \cdot \log(n) + \alpha_0$:

$$\mathsf{P}(s_1, s_2) = \mathsf{I}(a_1)\mathsf{I}(b_1)\mathsf{I}(a_2) \cdots \mathsf{I}(a_i)\mathsf{I}(\#)\mathsf{I}(b_2) \cdots \mathsf{I}(b_j)\mathsf{I}(\#)\mathsf{I}(\#) \cdots \mathsf{I}(\#)$$

where $\mathsf{I}(\mathtt{0}) = 10, \mathsf{I}(\mathtt{1}) = 11$ and $\mathsf{I}(\#) = 00$, in such a way that this word has length exactly $2(\alpha_1 \cdot \log(n) + \alpha_0 + 1)$. The "+1" origins from the extra character # which separates the two (tails of the) stacks. For convenience we use a typewriter font for the encoding I. Then, the encoding of stacks above is written

$$\mathsf{P}(s_1, s_2) = \mathtt{a_1}\,\mathtt{b_1}\,\mathtt{a_2}\,\mathtt{a_3} \cdots \mathtt{a_i}\,\mathtt{\#}\,\mathtt{b_2}\,\mathtt{b_3} \cdots \mathtt{b_j}\,\mathtt{\#}\,\mathtt{\#} \cdots \mathtt{\#}.$$

To perform the computations for some input of size n, we use a *configuration tree* which is a perfectly balanced tree of height $d + 2(\alpha_1 \cdot \log(n) + \alpha_0 + 1)$. It is used as a map from all[2] configurations to (some currently computed) values $\{\mathbf{0}, \mathbf{1}, \bot\}$. Given a configuration $K = (q, w_1, w_2)$, the leaf obtained following the path $q\mathsf{P}(w_1, w_2)$ from the root of the configuration tree is the stored value for that configuration. In other words, given a configuration tree t, the value corresponding to the configuration (q, s_1, s_2) is $\mathsf{d}_{q\mathsf{P}(s_1,s_2)}(t)$.

We describe now the process of the computation. The initial valued configuration tree has all leaves labeled by $\bot$ (this tree can be defined by explicit structural recursion, cf. Lemma 9). The strategy will be to update the leaves of the initial valued configuration tree, as many times as the running time of the machine. We will show that updates can be performed by a MIP-function. Then, we use Lemma 11 to iterate this update function. After this process, the output can be read on the left-most branch of the configuration tree, that is the path of the initial configuration $(q_0, \epsilon, \epsilon)$. So, to finish the proof, we have to show that such an update can be done by MIP-recursion.

For that, we introduce a function next which takes as input the currently computed valued configuration tree and the input tree. It returns the updated configuration tree. Actually, the function works by finite case distinction just calling auxiliary functions. By Lemma 6 next is shown MIP-definable[3]:

$$\mathsf{next}(\mathtt{t}_{d+4}, y) = \mathtt{t}_{d+4}[(x_{q\mathtt{ab}} \leftarrow \mathsf{next}_{q,a,b}(x_{q\mathtt{ab}}, \mathtt{t}_{d+4}, y))_{q \in \mathbb{W}_d, \mathtt{a}, \mathtt{b} \in \mathbb{W}_2}]$$

[2] Actually, all configurations with stacks smaller than $O(\log(n))$.

[3] Since, wrt the simulation, Equations for $m < d + 4$ play no role, we do not write them explicitly.

where $\mathsf{next}_{q,a,b}$ are the auxiliary functions. The role of these functions is to update the part of the configuration tree they correspond to. The definition of these auxiliary functions depends on the kind of states (accepting, rejecting, etc) and, for action states, on the top bits of the stacks.

We begin by the case of accepting and rejecting states. We define

$$\mathsf{next}_{q,a,b}(x,t,y) = \mathsf{const}_{\mathbf{1}}(x) \text{ if } q \text{ is accepting}$$
$$\mathsf{next}_{q,a,b}(x,t,y) = \mathsf{const}_{\mathbf{0}}(x) \text{ if } q \text{ is rejecting}$$

and use Lemma 7 to get MIP-definability.

For the *reading* states, we only provide the definition corresponding to a $1,1$-reading state. Other cases are similar, $\mathsf{next}_{q,a,b}(x,t,y) = \mathsf{read}(x, \mathsf{d}_a(y))$ with:

$$\mathsf{read}(\mathsf{t}_2, y) = (\mathsf{read}'(x_{00}, y) \star \mathsf{read}(x_{01}, y)) \star (\mathsf{read}(x_{10}, \mathsf{d}_{10}(y)) * \mathsf{read}(x_{11}, \mathsf{d}_{11}(y)))$$
$$\mathsf{read}'(x_0 \star x_1, y) = \mathsf{read}'(x_0, y) \star \mathsf{read}(x_1, y)$$
$$\mathsf{read}(c, y) = \bot$$
$$\mathsf{read}'(c, y) = \mathsf{cond}(y, \mathbf{0}, \mathbf{1}, \bot, \bot)$$

The hard cases are the action states. To compute the value of such configurations, we need the value of its two successor configurations. The key point is that the transitions of a configuration $(q, a_1 \cdots a_i, b_1 \cdots b_j)$ to its successors are entirely determined by the state q and the two top bits a_1 and b_1 so that $\mathsf{next}_{q,a_1,b_1}$ "knows" exactly which transition it must implement. We have to distinguish the four cases where we push or pop an element on one of the two stacks:

1. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{push}_1(\mathrm{a}_0))$;
2. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{pop}_1)$;
3. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{push}_2(\mathrm{b}_0))$;
4. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{pop}_2)$.

Let us see first how these action modify the encoding of configurations. So, we suppose the current configuration to be $K = (q, a_1 \cdots a_i, b_1 \cdots b_j)$. By assumption, the stacks of q' are the same as for q, so that the encoding of the first successor of K is

$$q' \, \mathtt{a}_1 \, \mathtt{b}_1 \, \mathtt{a}_2 \, \mathtt{a}_3 \cdots \mathtt{a}_i \, \# \, \mathtt{b}_2 \, \mathtt{b}_3 \cdots \mathtt{b}_j \, \# \# \cdots \#$$

For the second successor of K, the encoding depends on the four possible actions:

1. $q'' \, \mathtt{a}_0 \, \mathtt{b}_1 \, \mathtt{a}_1 \; \mathtt{a}_2 \; \mathtt{a}_3 \cdots \mathtt{a}_i \; \# \; \mathtt{b}_2 \; \mathtt{b}_3 \cdots \mathtt{b}_j \, \# \cdots \#$
2. $q'' \, \mathtt{a}_2 \, \mathtt{b}_1 \, \mathtt{a}_3 \cdots \; \mathtt{a}_i \; \# \; \mathtt{b}_2 \, \mathtt{b}_3 \cdots \mathtt{b}_j \; \# \; \# \, \# \cdots \#$
3. $q'' \, \mathtt{a}_1 \, \mathtt{b}_0 \, \mathtt{a}_2 \; \mathtt{a}_3 \cdots \; \mathtt{a}_i \; \# \, \mathtt{b}_1 \; \mathtt{b}_2 \; \mathtt{b}_3 \cdots \mathtt{b}_j \, \# \cdots \#$
4. $q'' \, \mathtt{a}_1 \, \mathtt{b}_2 \, \mathtt{a}_2 \; \mathtt{a}_3 \cdots \; \mathtt{a}_i \; \# \, \mathtt{b}_3 \cdots \mathtt{b}_j \; \# \; \# \, \# \cdots \#$

As for accepting and rejecting states, we will use auxiliary functions $\mathsf{next}_{\circ,1}$, $\mathsf{next}_{\circ,2,b_1}$, $\mathsf{next}_{\circ,3,b_1}$, and $\mathsf{next}_{\circ,4}$, which correspond to the four cases mentioned

above (and where $\circ$ is $\wedge$ or $\vee$ according to the state q). Then we use Lemma 7 to show the functions $\mathsf{next}_{q,a_1,b_1}$ defined by MIP-recursion.

We come back now to the definition of the four auxiliary functions $\mathsf{next}_{\circ,1}$, $\mathsf{next}_{\circ,2,b_1}$, $\mathsf{next}_{\circ,3,b_1}$, and $\mathsf{next}_{\circ,4}$. The principle of their definition is to follow in parallel the paths of the two successor configurations. To do that, we essentially use substitution of parameters, in the mutual in place recursion scheme.

For the case $\delta(q,\mathrm{a}_1,\mathrm{b}_1)=(q',q'',\mathsf{push}_1(\mathrm{a}_0))$, we define $\mathsf{next}_{q,a_1,b_1}(x,t,y)=\mathsf{next}_{\circ,1}(x,\mathsf{d}_{q'a_1b_1}(t),\mathsf{d}_{q''a_0b_1a_1}(t))$. With respect to the configuration tree encoding and to the definition of next, observe that $\mathsf{next}_{\circ,1}(x,u,v)$ is fed with the arguments $(\mathsf{d}_{qa_1b_1}(t),\mathsf{d}_{q'a_1b_1}(t),\mathsf{d}_{qa_0b_1a_1}(t))$ where t is the configuration tree to be updated. So that the height of the last argument is two[4] less than the others. In this case, we can go in parallel, with the only *previso* that the second stack is shorter. Equations below cope with that technical point. Formally we define $\mathsf{next}_{\circ,1}$ as:

$$\begin{aligned}
\mathsf{next}_{\circ,1}(\mathfrak{t}_2,u,v)&=\mathfrak{t}_2[(x_w\leftarrow\mathsf{next}_{\circ,1'}(x_w,\mathsf{d}_w(u),\mathsf{d}_w(v)))_{w\in\mathbb{W}_2}]\\
\mathsf{next}_{\circ,1'}(\mathfrak{t}_4,u,v)&=\mathfrak{t}_4[(x_w\leftarrow\mathsf{next}_{\circ,1'}(x_w,\mathsf{d}_w(u),\mathsf{d}_w(v)))_{w\in\mathbb{W}_4}]\\
\mathsf{next}_{\circ,1'}(\mathfrak{t}_2[x_w\leftarrow c_w],u,v)&=\mathfrak{t}_2[(x_w\leftarrow\mathsf{d}_w(u)\circ v)_{w\in\mathbb{W}_2}]
\end{aligned}$$

where the c_w are to be taken in $\{\mathbf{0},\mathbf{1},\bot\}$ and $\circ$ is the conditional corresponding to the state.

If $\delta(q,\mathrm{a}_1,\mathrm{b}_1)=(q',q'',\mathsf{pop}_1)$, let $\mathsf{next}_{q,\mathrm{a}_1,\mathrm{b}_1}(x,t,y)=\mathsf{next}_{\circ,2,\mathrm{b}_1}(x,\mathsf{d}_{q'a_1b_1}(t),\mathsf{d}_{q^{\shortparallel}(t)})$. In that case, it is the last argument which is the bigger one.

$$\begin{aligned}
\mathsf{next}_{\circ,2,b_1}(\mathfrak{t}_2,u,v)&=\mathfrak{t}_2[(x_w\leftarrow\mathsf{next}'_{\circ,2}(x_w,\mathsf{d}_w(u),\mathsf{d}_{wb_1}(v)))_{w\in\mathbb{W}_2}]\\
\mathsf{next}'_{\circ,2}(\mathfrak{t}_2,u,v)&=\mathfrak{t}_2[(x_w\leftarrow\mathsf{next}'_{\circ,2}(x_w,\mathsf{d}_w(u),\mathsf{d}_w(v)))_{w\in\mathbb{W}_2}]\\
\mathsf{next}'_{\circ,2}(c,u,v)&=u\circ\mathsf{d}_{00}(v)
\end{aligned}$$

If $\delta(q,\mathrm{a}_1,\mathrm{b}_1)=(q',q'',\mathsf{push}_2(\mathrm{b}_0))$, we define $\mathsf{next}_{q,\mathrm{a}_1,\mathrm{b}_1}$ by the equation:

$$\begin{aligned}
\mathsf{next}_{q,\mathrm{a}_1,\mathrm{b}_1}(x,t,y)&=\mathsf{next}_{\circ,3,\mathrm{b}_1}(x,\mathsf{d}_{q'a_1b_1}(t),\mathsf{d}_{q^{\shortparallel}a_1b_0(t)})\\
\mathsf{next}_{\circ,3,b_1}(\mathfrak{t}_2,u,v)&=(\mathsf{next}_{\circ,1}(x_{00},\mathsf{d}_{00}(u),\mathsf{d}_{00b_1}(v))\star\mathsf{next}_{\circ,1}(x_{01},\mathsf{d}_{01}(u),\mathsf{d}_{01b_1}(v)))\star\\
&\qquad(\mathsf{next}_{\circ,3,b_1}(x_{10},\mathsf{d}_{10}(u),\mathsf{d}_{10}(v))\star\mathsf{next}_{\circ,3,b_1}(x_{11},\mathsf{d}_{11}(u),\mathsf{d}_{11}(v)))\\
\mathsf{next}_{\circ,3,b_1}(c,u,v)&=\bot
\end{aligned}$$

For the last case, that is $\delta(q,\mathrm{a}_1,\mathrm{b}_1)=(q',q'',\mathsf{pop}_2)$, we use four auxiliary arguments to remind the first letter read on the stack of the second successor.

$$\begin{aligned}
\mathsf{next}_{q,\mathrm{a}_1,\mathrm{b}_1}(x,t,y)&=\mathsf{next}_{\circ,4,\epsilon}(x,\mathsf{d}_{q'a_1b_1}(t),\mathsf{d}_{q^{\shortparallel}00}(t),\mathsf{d}_{q^{\shortparallel}01}(t),\mathsf{d}_{q^{\shortparallel}10}(t),\mathsf{d}_{q^{\shortparallel}11}(t))\\
\mathsf{next}_{\circ,4,00}(\mathfrak{t}_2,u,v_{00},v_{01},v_{10},v_{11})&=\mathfrak{t}_2[(x_w\leftarrow\mathsf{next}'_{\circ,2}(x_w,\mathsf{d}_w(u),v_w))_{w\in\mathbb{W}_2}]\\
\mathsf{next}_{\circ,4,v}(\mathfrak{t}_2,u,v_{00},v_{01},v_{10},v_{11})&=\mathfrak{t}_2[(x_w\leftarrow\mathsf{next}_{\circ,4,w}(x_w,\mathsf{d}_w(u),\mathsf{d}_w(v_{00}),\\
&\qquad\qquad\mathsf{d}_w(v_{01}),\mathsf{d}_w(v_{10}),\mathsf{d}_w(v_{11}))_{w\in\mathbb{W}_2}]\\
\mathsf{next}_{\circ,4,v'}(c,u,v_{00},v_{01},v_{10},v_{11})&=\bot
\end{aligned}$$

with $v\in\{\epsilon,01,10,11\}$ and $v'\in\mathbb{W}_0\cup\mathbb{W}_2$.

[4] one bit of the stack is encoded as two bits in the configuration tree.

5 Compilation of recursive definitions to circuit

This section is devoted to the proof of the Proposition:

Proposition 14. *For $k \geq 1$, any function in INC^k is computable in NC^k.*

We begin with some observations. All along, n denotes the size of the input. First, to simulate theoretic functions in INC^k, we will forget the tree structure and make the computations on the words made by the leaves. Actually, since the trees are always full balanced binary trees, we could restrict our attention to input of size 2^k for some k.

Second, functions defined by explicit structural recursion can be computed by NC^1 circuits. This is a direct consequence of the fact that explicit structural recursion is a particular case of LRRS-recursion as defined in Leivant and Marion [14].

Third, by induction on the definition of functions, one proves the key Lemma:

Lemma 15. *Given a function $f \in \mathsf{INC}^k$, there are (finitely many)* MIP*-functions $h_1, \ldots, h_m$ and polynomials $P_1, \ldots, P_m$ of degree smaller than k such that $f(\bar{t}, \bar{u}) = h_1^{P_1(\log(n))}(\cdots h_m^{P_m(\log(n))}(g(\bar{u}))\ldots)$ where g is defined by structural recursion.*

Now, the compilation of functions to circuits relies on three main ingredients. First point, we show that each function h_i as above can be computed by a circuit:

1. of fixed height with respect to the input (the height depends only on the definition of the functions),
2. with a linear number of gates with respect to the size of the first input of the circuit (corresponding to the recurrence argument),
3. with the number of output bits equal to the number of input bits of its first argument.

According to 1), we note H the maximal height of the circuits corresponding to the h_i's.

Second point, since there are $\sum_{i=1..m} P_i(\log(n))$ applications of such h_i, we get a circuit of height bounded by $H \times \sum_{i=1..m} P_i(\log(n)) = O(\log^k(n))$. That is a circuit of height compatible with NC^k. Observe that we have to add as a first layer a circuit that computes g. According to our second remark, this circuit has a height bounded by $O(log(n))$, so that the height of the whole circuit is of the order $O(\log^k(n))$.

Third point, the circuits corresponding to g, being in NC^1, have a polynomial number of gates with respect to n and a polynomial number of output bits with respect to n. Observe that the output of g is exactly the recurrence argument of some h_i whose output is itself the first argument of the next h_i, and so on. So that according to item 3) of the first point, the size of the input argument of each of the h_i is exactly the size of the output of g. Consequently, according to item 2) above, the number of circuit gates is polynomial.

Since all constructions are uniform, we get the expected result.

5.1 NC^0 circuits for mutual recursion

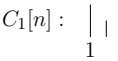
In this section, we prove that functions defined by mutual in place recursion can be computed by NC^0 circuits with a linear number of gates wrt the size of the first argument. Since MIP-functions keep the shape of their first argument, we essentially have to build a circuit for each bit of this argument.

Lemma 16. *Explicitely defined boolean functions can be defined without use of $\star$.*

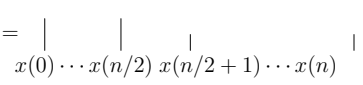
Lemma 17. *Explicitly defined boolean functions are in NC^0.*

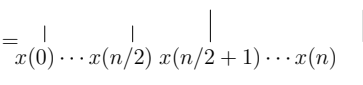
Proof. Consider the following circuits. To stress the fact that circuits are uniform, we put the size of the arguments into the brackets. n corresponds to the size of x, n_0 to the size of x_0 and so on. $x(k)$ for $k \in \mathbb{N}$ corresponds to the k-th bit of the input x. The "long" wires correspond to the outputs. Shorter ones are simply forgotten.

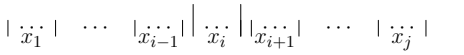
$C_0[n]$: 

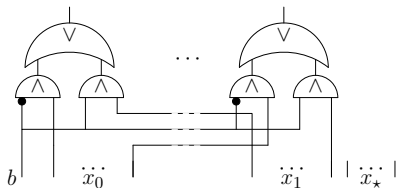
$C_1[n]$: 

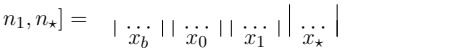
$C_{\mathtt{d}_0}[1] = C_{\mathtt{d}_1}[1] =$ 

$C_{\mathtt{d}_0}[2+n] =$ 

$C_{\mathtt{d}_1}[2+n] =$ 

$C_{\pi_i^j}[n_1, \cdots, n_j]$: 

$C_{\mathtt{cond}}[1, n_0, n_0, n_\star] =$ 

$C_{\mathrm{cond}}[2+n_b, n_0, n_1, n_\star] =$ 

We see that composing the previous cells, with help of Lemma 16, we can build a circuit of fixed height (wrt to the size of input) for any explicitly defined boolean function. Observe that the constructions are clearly uniform.

5.2 Simulation of time recursion

Lemma 18. *Any MIP-function can be computed by a circuit of fixed height wrt the size of the input.*

Proof. Let us consider a set $(f_i)_{i\in I}$ of MIP-functions. Write their equations as follows:

$$f_i(t_0 \star t_1, \bar{u}) = f_{p(i,0)}(t_0, \bar{\sigma}_{i,0}(\bar{u})) \star f_{p(i,1)}(t_1, \bar{\sigma}_{i,1}(\bar{u}))$$
$$f_i(c, \bar{u}) = g_i(c, \bar{u})$$

where $p(i,b) \in I$ is an explicit (finite) mapping of the indices, $\bar{\sigma}_{i,0}$ and $\bar{\sigma}_{i,1}$ are vectors of $\star$-free explicitely defined functions and the functions $g_{i,c}$ (and consequently the g_i) are explicitly defined boolean functions.

First, observe that any of these explicitly defined functions g_i can be computed by some circuit B_i of fixed height as seen in Lemma 17. Since I is finite, we call M the maximal height of these circuits $(B_i)_{i\in I}$.

Suppose we want to compute $f_i(t, \bar{x})$ for some t and $\bar{x}$ which have both size smaller than n. Remember that the shape of the output is exactly the shape of the recurrence argument t. So, to any k-th bit of the recurrence argument t, we will associate a circuit computing the corresponding output bit, call this circuit C_k. Actually, we will take for each k, $C_k \in \{B_i : i \in I\}$. Putting all the circuits $(C_k)_k$ in parallel, we get a circuit that computes all the output bits of f_i, and moreover, this circuit has a height bounded by M. So, the last point is to show that for each k, we may compute uniformly the index i of the circuit B_i corresponding to C_k and the inputs of the circuit C_k.

To denote the k-th bit of the input, consider its binary encoding where we take the path in the full binary tree t ending at this k-th bit. Call this path w. Notice first that w itself has logarithmic size wrt n, the size of t. Next, observe that any sub-tree of the inputs can be represented in logarithmic size by means of its path. Since all along the computations, the arguments $\bar{u}$ are sub-trees of the input, we can accordingly represent them within the space bound.

To represent the value of a subterm of some input, we use the following data structure. Consider the record type $\mathtt{st} = \{\mathtt{r}; \mathtt{w}; \mathtt{h}\}$. The field `r` says to which input the value corresponds to. `r = 0` corresponds to t, `r = 1` correspond to x_1 and so on. `w` gives the path to the value (in that input). For convenience, we keep its height `h`. In summary `{r=i;w=w';h=m}` corresponds to the subtree $\mathtt{d}_{w'}(u_i)$ (where we take the convention that $t = u_0$). We use the '.' notation to refer to a field of a record. We consider then the data structure $\mathtt{val} = \mathtt{st} + \{0, 1\}$. Variables `u, v` coming next will be of that "type".

To compute the function $(\sigma_{i,b})_{i\in I, b\in\{0,1\}}$ appearing in the definition of the $(f_i)_{i\in I}$, we compose the programs:

```
zero(u){
  return 0;
}
```

```
one(u){
  return 1;
}
```

```
pi_i_j(u_1,...u_j){
  return u_i;
 }
```

```
d0(u){ if(u == 0 || u == 1 || u.h = 0)  return u;
         else  return [r=u.r;w=u.w 0;h= u.h-1];   }

d1(u){ if(u == 0 || u == 1 || u.h == 0) return u;
         else return [r=u.r;w=u.w 1;h= u.h-1];    }

cond(u_b,u_0,u_1,u_s){
  if (u_b == 0 ||
     (u_b.h == 0 && last-bit(u_b.w) == 0))
     return u_0;
  elseif (u_b == 1||
         (u_b.h == 0 && last-bit(u_b.w) == 1))
             return u_1;
  else  return u_s;
}
```

Then we compute the values of i and the $\bar{u}$ in $g_i(c, \bar{u})$ corresponding to the computation of the k-th bits of the output. Take $d + 1$ the maximal arity of functions in $(f_i)_{i \in I}$. To simplify the writing, we take it (wlog) as a common arity for all functions.

```
G(i,w,u_0,...,u_d){
 //u_0 corresponds to t,
   if(w == epsilon) {
   return(i,u_0,...,u_d);
   }
   else{
   a := pop(w); //get the first letter of w
   w := tail(w); //remove the first letter to w
   switch(i,a){//i in I, a in {0,1}
         case (i1,0):
         v_0 = d_0(u_0);
         foreach 1 <= k <= d:
               v_k = sigma_i1_0_k(u_0,...,u_d);
               //use the sigma defined above
               next_i = p_i1_0;
               //the map p is hard-encoded
         break;
         ...
         case (im,1):
         v_0 = d_1(u_0);
         foreach 1 <= k <= d:
               v_k = sigma_im_1_k(u_0,...,u_d);
               next_i = p_im_1;
         break;
        }
        return G(next_i,w,d_a(u_0),v_1,...,v_d);
   }
}
```

Observe that this program is a tail recursive program. As a consequence, to compute it, one needs only to store the recurrence arguments, that is a finite number of variables. Since the value of these latter variables can be stored in logarithmic space, the computation itself can be performed within the bound. Finally, the program returns the name `i` of the circuit that must be build, a pointer on each of the inputs of the circuit with their size. It is then routine to build the corresponding circuit at the corresponding position `w`.

References

1. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.
2. J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural complexity II*, volume 22 of *EATCS Monographs of Theoretical Computer Science*. Springer, 1990.
3. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
4. S. Bellantoni and I. Oitavem. Separating NC along the δ axis. *Theoretical Computer Science*, 318:57–78, 2004.
5. G. Bonfante, R. Kahle, J.-Y. Marion, and I. Oitavem. Towards an implicit characterization of NC^k. In Zoltán Èsik, editor, *Computer Science Logic '06*, volume 4207 of *Lecture Notes in Computer Science*, pages 212–224. Springer, 2006.
6. G. Bonfante, J.-Y. Marion, and R. Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR*, pages 90–104, 2006.
7. A. K. Chandra, D. J. Kožen, and L. J. Stockmeyer. Alternation. *Journal ACM*, 28:114–133, 1981.
8. J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.
9. Neil Immerman. *Descriptive Complexity*. Springer, 1998.
10. L. Kristiansen and N. D. Jones. The flow of data and the complexity of algorithms. In S. B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2005.
11. D. Leivant. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991.
12. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.
13. D. Leivant. A characterization of NC by tree recurrence. In *Foundations of Computer Science 1998*, pages 716–724. IEEE Computer Society, 1998.
14. D. Leivant and J.-Y. Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236(1–2):192–208, 2000.
15. J.-Y. Marion. Predicative analysis of feasibility and diagonalization. In *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, 2007.
16. K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.
17. I. Oitavem. Characterizing *NC* with tier 0 pointers. *Mathematical Logic Quarterly*, 50:9–17, 2004.
18. W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.
19. H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.